DOCUMENT PREPARATION SYSTEMS
J. Nievergelt, G. Coray, J.D. Nicoud, A.C. Shaw (eds.)
© North-Holland Publishing Company, 1982

UNIX Document Preparation*

Brian W. Kernighan and Michael E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The UNIX† operating system provides programs for sophisticated document preparation within the framework of a general-purpose operating system. The document preparation software includes text editors, programmable text formatters, macro-definition packages for a variety of page layout styles, special processors for mathematical expressions, tabular material, bibliographic references, and line drawing, and numerous supporting programs, such as a spelling-mistake detector.

In practice, this collection of facilities has proven to be easy to learn and use, even by secretaries, typists, and other non-specialists. Experiments have shown that preparation of complicated documents is about twice as fast as on completely manual systems. There are many benefits to using a general-purpose operating system instead of specialized stand-alone terminals or a system dedicated to "word processing." On the UNIX system, these include an excellent software development facility and the ability to share computing and data resources among a community of users.

1. INTRODUCTION

We use the term *document preparation* to mean the creation, modification, and display of textual material, such as manuals, reports, papers, and books. "Document preparation" seems preferable to "text processing" (which is not precise), or "word processing" (which has connotations of stand-alone specialized terminals).

A variety of comparatively elegant output devices has become available, supplementing the traditional typewriters, terminals, and line printers; this has led to a much increased interest in automated document preparation. Computerized systems are no longer limited to straight text printed in unattractive constant-width characters, but can produce a full range of documents in attractive fonts and page layouts. The major example of an output device with significant capabilities is the phototypesetter, which produces high quality printed output on photographic paper or film. Other devices include typewriter-like terminals capable of high-resolution motion, dot matrix printer-plotters, microfilm recorders, and xerographic printers.

Some of the benefits of computer-aided document preparation are familiar. Text is only entered once, and editing is only needed where changes are being made. There are many other benefits, however. Special languages can be used to simplify the entry of complex material such as tables and mathematical expressions. The format of a document can be decoupled from its content;

† UNIX is a Trademark of Bell Laboratories.

* This is a revised version of a paper entitled "Document Preparation" that appeared in the Bell System Technical Journal, vol. 57, no. 6. pp. 2115-2135, August 1978. Reprinted by permission.

the only format-control information that need be embedded is that describing textual categories and boundaries, such as titles, section headings, and paragraphs. Alternative document styles are then possible through the use of different formatting programs and different interpretations applied to the embedded format controls. Furthermore, programs can examine text to detect spelling mistakes, compare versions of documents, and prepare indexes automatically. Machine-generated data can be incorporated in documents; excerpts from documents can be fed to programs without transcription.

Further advantages accrue when document preparation is done on a general-purpose computer system. One is the sharing of programs and data bases among users; programs originally written for some other purpose may be useful to the document preparer. Having a broad range of users, from typists to scientists, on the same system leads to an unusual degree of cooperation in the preparation of documents. In addition, the existence of electronic mail and inter-machine networks makes possible the transmission of documents in machine-readable form at electronic speed.

The UNIX document preparation software includes several text editors. All can be used on documents, data files, or programs. The most often used are the original editor, **ed**, which is a line-oriented command-driven editor, and **vi**, a cursor-driven video editor. Two programmable text formatters, **nroff** and **troff**, provide paginated formatting and allow unusual freedom and flexibility in determining the style of documents. Augmented by various macro-definition packages, **nroff** and **troff** provide footnote processing, multiple-column output, column-length balancing, and automatic figure placement. A preprocessor called **eqn** translates a simple language for describing mathematical expressions into formatter input; a table-construction preprocessor, **tbl**, provides an analogous facility for input of data and text that is to be arranged into tables. Similarly **refer** extracts bibliographic references from a common data base and inserts them, in various formats, into documents. Most recently, the **pic** and **ideal** languages permit the typesetting of line drawings.

Besides describing these tools in some detail, we will also mention some supporting software useful to the document preparer, and summarize some comparisons between manual methods of document preparation and methods using UNIX.

2. TEXT EDITING

The original UNIX text editor was **ed**. We will not try to give a complete description of **ed** here; details may be found in Ref. 1. Rather, we will try to mention those attributes that are most interesting and unusual.

The **ed** editor is not specialized to any kind or style of text; it is used for programs, data, and documents alike. It is based on editing commands such as "print" and "substitute," rather than on special function keys, and provides convenient facilities for selecting the text lines to be operated on and altering their contents. Since it does not use special function keys or cursor controls, it does not require a particular kind of input device. It has a relatively restricted command set, with no macros or programming capabilities.

A text editor is often the primary interface between a user and the system, and the program with which most user time is spent. Accordingly, an editor has to be easy to use, and efficient of the user's time—editing commands have to "flow off the fingertips." In accordance with this principle, **ed** is quite terse. Each editor command is a single letter, e.g., **p** for "print," and **d** for "delete." Most commands may be preceded by zero, one, or two "line addresses" to affect, respectively, the "current line" (i.e., the line most recently referenced), the addressed line, or the range of contiguous lines between and including the pair of addresses. There are also shorthands for the current line and the last line of the file. Arithmetic expressions involving line numbers are also permitted:

```
-5,+5p
```

prints from five lines before the current line to five lines after, while

```
$-5,$p
```

prints the last six lines. In both cases, the last line printed becomes the current line, so that subsequent editing operations may begin from there.

Most often, the lines to be affected are specified not by line number, but by context, that is, by naming some text pattern that occurs in them. The "line address"

```
/abc/
```

refers to the first line after the current line that contains the pattern **abc**. This line address standing by itself will find and print the next line that contains **abc**, while

```
/abc/d
```

finds it and deletes it. Context searches begin with the line immediately after the current line, and wrap around from the end of the file to the beginning if necessary. It is also possible to scan the file in the reverse direction by enclosing the pattern in question marks: **?abc?** finds the previous **abc**.

The substitute command **s** can replace any pattern by any literal string of characters in any group of lines. The command

```
s/ofrmat/format/
```

changes **ofrmat** to **format** on the current line, while

```
1,$s/ofrmat/format/
```

changes it on every line. In both searches and substitute commands, the pattern // is an abbreviation for the most recently used pattern, and **&** stands for the most recently matched text. Both can be used to avoid repetitive typing. The "undo" command **u** undoes the most recent substitution.

In all cases, the trailing delimiter ("/", for instance) may be omitted if it is the last thing in the command.

Text can be added before or after any line, and any group of contiguous lines may be replaced by new lines. "Cut and paste" operations are also possible—any group of lines may be either moved or copied elsewhere. Individual lines may be split or joined together; text within a line may be rearranged.

The editor does not work on a file directly, but on a copy. Any file may be read into the working text at any point; any contiguous lines may be written or appended to any file. And any UNIX command may be executed from within the editor, even another instance of the editor.

So far, we have described the basic editor features: this is all that the beginning user needs to know, and about as much as most of our clerical users need. More sophisticated users can include in their search patterns "metacharacters" that specify character classes, repetition of characters or classes, the beginning or end of a line, and so on. For example, the pattern

```
/^[0-9]/
```

searches for the next line that begins with a digit.

Any set of editing commands may be done under control of a "global" command: the editing commands are performed starting at each line that matches a pattern specified in the global command. As the simplest example,

```
g/interesting/p
```

prints all lines that contain **interesting**.

Finally, given the UNIX software for input-output redirection, it is easy to make a "script" of editing commands in advance, then run it on a sequence of files.

The basic pattern-searching and editing capabilities of **ed** have been co-opted into other, more specialized programs as well. The program **grep** ("global regular expression print") prints all input lines that contain a specified pattern; this program is particularly useful for finding the location of an item in a set of files, or for culling items from larger inputs. The program **sed** is a variant of **ed**

that performs a set of editing operations on each line of an input stream

Some users prefer more complex editors. The **vi** editor, written by Bill Joy at Berkeley, depends on cursor motion and so runs only on video terminals. It maintains a current screen display of the file at all times. The 'undo' feature of this editor is very attractive for learning; it can back up across any command and restore the previous version of the text. **vi** is not nearly as heavily used as **ed**, partly because it simply is not as familiar, and partly because it places a significantly greater load on the system.

The **emacs** editor, originated by Richard Stallman of MIT, is another screen editor in limited use, but it is too complex for the typical typist, at least as normally presented. **emacs** is programmable, however, so users can define their own editing interfaces if they wish.

As an operating system, UNIX provides sufficient facilities for these editors and indeed a variety of others; the system does not foreclose any options. The choice of which one to use is thus made according to personal taste, although there is some pressure to mimimize the number of different editors so as to avoid problems of maintenance and training. It is easier for users to help each other if they are all using basically the same tools.

3. TROFF AND NROFF — BASIC TEXT FORMATTERS

Once a user has entered a document into the file system, it can be formatted and printed by **troff** or **nroff**.[2] These are programmable text formatters that accommodate a wide variety of formatting tasks by providing flexible fundamental tools rather than specific features. **troff** supports phototypesetting on any typesetter; we use equipment from Wang, Mergenthaler, and Autologic today. **nroff** produces formatted output for a variety of terminals and line printers, using the full capabilities and resolution of each. **troff** and **nroff** accept the same input language. Except for device description tables, device-oriented routines, and a small amount of conditionally-compiled code, the source code for these programs is also identical. Both programs are driven by device description tables, so that **troff** can drive a variety of typesetters and **nroff** can understand the entire typesetter character set, printing non-ASCII characters where available or where they can be constructed (by overstriking) on a particular device. The remaining discussion in this section focuses on **troff**; the behavior of **nroff** is identical within device capability.

troff is intended to permit unusual freedom in user-designed document styles, at the price of being somewhat hard to use for basic formatting tasks. The fundamental operations that **troff** provides are sufficient for programming complicated formatting tasks. For example, footnote processing, multi-column layout with column balancing, and automatic figure placement are not built-in operations, but are programmed in **troff** macros when needed. To program in **troff**, the user writes a set of macro definitions, which expand into the command sequences needed for each formatting task. **troff** may also be instructed to invoke specified macros automatically at particular page positions, such as at the top and near the bottom of the page; other commands are invoked by the user by placing macro calls at paragraphs, section headings, and other relevant boundaries. Once a macro package has been written for some particular style of document, users preparing a document in that style need only prepare their text with macro calls at the appropriate points, as described in the next section.

Complex formatting tasks require complex macro packages designed by competent programmers. A well-designed package can be easy to use, and usually permits convenient choice between several related styles. At the simple end of the style spectrum, a newspaper-style galley may not require any embedded format control except paragraphing. A simple paginated style might use only three macros, defining the nature of the top-of-page margin, the bottom-of-page margin, and paragraph breaks.

Input consists of text lines, which are destined to be printed, interspersed with control lines, which set parameters or otherwise control subsequent processing. Control lines begin with a control character—normally a period—followed by a one- or two-character name that specifies either a built-in request or the substitution of a user-defined macro in place of the control line. This form is reminiscent of earlier text formatters.[3,4] A typical sequence of requests is

```
.pl 8.5i
.ll 5i
.ps 10
.ft R
.sp 2
```

which sets the page length to 8.5 inches, the line length to 5 inches, the type size to 10 points, the font to Roman, and then spaces down two lines.

Formatting functions may also be introduced anywhere in the input by an in-line command, which consists of an escape character, normally \, a command, and perhaps parameters. For example, **f** specifies a font change, so **fB** causes a change to bold font when it appears, and **fP** reverts to the previous font. Similarly, **l´3i´** draws a three-inch line.

There are about eighty built-in control-line requests that implement the fundamental operations, allow the setting of parameters, and otherwise affect format control. In addition, more than forty in-line commands may appear anywhere to specify certain characters, to set indicators, and to introduce various functions. Automatic services available include filling and adjusting of text, hyphenation with user control over exceptions, user-settable left, right, and centering tabs, and output line numbering.

User-settable parameters include font, point size, page length, page number, line spacing, line length, indent, and tabs. Functions are available for building brackets, overstriking, drawing lines, circles, ellipses, arcs and splines, generating vertical and horizontal motions, and calculating the width and height of a string. In addition to the parameters that are defined by the formatter, users may define their own, stored in **troff** variables called "number registers" (for numeric parameters) and "strings" (for character data). These variables may be used in arithmetic and logical expressions to set parameters or to control the invocation of macros or requests.

A *macro* is a user-named set of lines of arbitrary text and format control information. It is interpolated into the input stream either by invoking it by name, or by specifying that it is to be invoked when a particular vertical position on a page is reached. Arguments may be passed to a macro invoked by name. A *string* is a named string of characters that may be interpolated at any point. Macros and strings may be created, redefined, appended to, renamed, and removed. Macros may be nested to an arbitrary depth, limited only by the memory available.

Processed text may be diverted into a macro instead of being output, for footnote collection or to determine the horizontal or vertical size of a block of text before final placement on a page. When reread, diverted text retains its character fonts and sizes and overall dimensions.

A *trap* mechanism provides for action when certain conditions occur. The conditions available are position on the current output page, length of a diversion, and input line count. A macro associated with a vertical page position is automatically invoked when a line of output falls on or after the trap position. For example, reaching a specified place near the bottom of the page could invoke a macro that describes the bottom margin area. Similarly, a vertical position trap may be specified for diverted output. An input line count trap causes a macro to be invoked after a specified number of input text lines have been read.

Some parameters are available to the user in predefined number registers; these include the day, month, year, page number, input line number, etc. In addition, users may define their own registers. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in decimal, alphabetic, or roman numeral format. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired. In most circumstances, numerical input may have appended scale factors representing inches, points, ems, etc. Numerical input may be provided by expressions involving a variety of arithmetic and logical operators.

A mechanism is provided for conditionally accepting a group of lines as input. The conditions that may be tested are the value of a numerical expression, the equality of two strings, and the truth of built-in conditions such as page number parity or whether the formatter is **troff** or **nroff**.

Certain of the parameters that control text processing constitute an *environment*, which may be switched by the user. It is convenient, for example, to process footnotes in a separate environment from the main text. Environment parameters include line length, line spacing, indent, and character size and font. In addition, any collected but not yet output lines or words are a part of the environment. Parameters that are global and not switched with the environment include, for example, page length, page position, and macro, string and number register values.

It is not possible to give any substantial examples of **troff** macro definitions, but we will sketch a few to indicate the general style of use.

The simplest example is to provide pagination—an extra space at the top and bottom of each page. Two macros are usually defined—a *header* macro containing the top-of-page text and spacings, and a *footer* macro containing the bottom-of-page text and spacings. A trap must be placed at vertical position zero to cause the header macro to be invoked and a second trap must be placed at the desired distance from the bottom for the footer. Simple macros merely providing space for the margins could be defined as follows.

```
.de hd          \"  begin header definition
'sp 1i          \"  space 1 inch
. .             \"  end of header definition
.de fo          \"  footer
'bp             \"  space to beginning of next page
. .             \"  end of footer definition
.wh 0 hd        \"  set trap to invoke hd when at top of page
.wh -1i fo      \"  set trap to invoke fo 1 inch from bottom
```

Normally the **.sp** and **.bp** commands cause a *break*, that is, the current output line is flushed and a new one begun. A command that begins with a quote (as in **'bp**) does not cause a break when it is invoked. The sequence \" introduces a **troff** comment.

A more complicated header might print the current page number at the top right or left, being sure to preserve the proper size and font:

```
.de hd
'sp 0.5i
.if o .tl '''\\s10\\fR- \n% -\\s0\\fP' \" odd page
.if e .tl '\\s10\\fR- \n% -\\s0\\fP''' \" even page
'sp 0.5i
..
```

The backslashes are necessary inside a macro to delay evaluation until the macro is invoked.

The production of multi-column pages requires still more complicated macros. The basic idea is that the header macro records the vertical position of the column top in a register and initializes a column counter. The footer macro is invoked at the bottom of each column. Normally it increments the column counter, increments the page offset by the column width plus the column separation, and generates a reverse vertical motion to the top of the next column (the place recorded by the header macro). After the last column, however, the page offset is restored and the bottom margin functions for the complete page occur.

Footnote processing is even more involved; only the general strategy will be summarized here. A pair of macros is defined that allows the user to indicate the beginning and end of the footnote text. The footnote-start macro changes to the footnote environment and begins a diversion that appends to a macro in which footnotes are being collected. The footnote-end macro terminates the diversion, resets the environment, and moves the footer trap up the page an amount equal to the size of the diverted footnote text. The footer eventually invokes and then removes the macro containing the accumulated footnotes and resets its own trap position. Footnotes that don't fit have their overflow rediverted and are treated as the beginning footnote on the next page.

The use of preprocessors to convert special input languages for equations and tables into **troff** input means that many documents reach **troff** containing large amounts of program-generated

input. For example, a simple equation might produce dozens of **troff** input lines and require many string definitions, redefinitions, and detailed numerical computations for proper character positioning. The **troff** string that finally contains the equation contains many font and size changes and local motions, and so can become very long. All of this demands substantial string storage, efficient storage allocation, larger text buffers than would otherwise be necessary, and the accommodation of large numbers of strings and number registers. Input generated by programs instead of people severely tests program robustness.

Regardless of the input, the output of **troff** is a simple, typesetter-independent language that basically tells where to put each character on the page, and in what size and font. A post-processor interprets this language and generates the necessary commands to drive a particular output device. There is generally one post-processor per typesetter.

4. MACROS—DECOUPLING CONTENT AND FORMAT

Although **troff** provides full control over typesetter (or terminal) features, few users exercise this control directly. Just as programmers have learned to use problem-oriented languages rather than assembly languages, it has proven better for people who prepare documents to describe them in terms of content, rather than specifying point sizes, fonts, etc., in a typesetter-oriented way. This is done by avoiding the detailed commands of **troff**, and instead embedding in the text only macro commands that expand into **troff** commands to implement a desired format.

For example, the title of a document might be prefaced by

```
.TL
```

which would expand, for this paper, into "Times Roman bold, 11 point type, centered at top of new page," but for a journal might be "Helvetica Bold, 14 point, left adjusted, preceded by copyright notice and a one-inch space," or whatever is desired. In a similar way, there would be macros for other common features of a document, such as author's name, abstract, section, paragraph, and footnote.

This document actually begins

```
.TL
UNIX Document Preparation*
.AU
Brian W. Kernighan and Michael E. Lesk
.AI
.MH
.AB
.PP
The
.UX
operating system provides programs
for sophisticated document preparation ...
.PP
In practice, this collection of facilities has proven to be
easy to learn and use ...
.AE
.FS
* This is a revised version of a paper ...
.FE
```

Some of these macros are obviously specialized to our environment. For instance, **.MH** expands into the company name and address, and **.UX** becomes UNIX, with a footnote about trademark on the first occurrence.

Macro packages have been prepared for a variety of document styles. Locally, these include formal and informal internal memoranda; technical reports for external distribution; the Association for Computing Machinery journals; some American Physical Society journals; *Software—Practice and Experience*; and *The Bell System Technical Journal*. All of these macro packages recognize standard

macro names for titles, paragraphs, and other document features, so the same input can be made to appear in many different forms, without changing it at all. Authors can prepare camera-ready copy for any of these journals, merely by naming the journal type when **troff** is invoked.

An important advantage of this system is the ease with which new users learn document preparation. It is necessary only to learn the correct way to describe document content and boundaries, not how to control the typesetter at a detailed level. A typist can easily learn the dozen or so most common macros in a few minutes, and another dozen as needed. This entire article uses only about 30 distinct macro calls, rather more than the norm.

Although **nroff** is used for typewriter-like output, and **troff** for photocomposition, they accept exactly the same input language, and thus hide details of particular devices from users. Macro packages also provide a degree of independence: they permit a uniformity of *input*, so that input documents look the same regardless of the output format or device they eventually appear in. This means that to find the title of a document, for example, it is not necessary to know what format is being used to print it. Finally, macros also enforce a uniformity of *output*. Since each output format is defined in appearance by the macro package that generates it, all documents prepared in that format will look the same.

5. EQN—A PREPROCESSOR FOR MATHEMATICAL EXPRESSIONS

Much of the work of Bell Laboratories is described in technical reports and papers containing significant amounts of mathematics. Mathematical material is difficult to type and expensive to typeset by traditional methods. Because of positioning requirements and the multiplicity of characters, sizes, and fonts, it is not feasible for a human to typeset mathematics directly with **troff** commands. **troff** has the facilities needed for preparing mathematical expressions, such as arbitrary horizontal and vertical motions, line-drawing, size changing, etc., but it is not easy to use them directly because of the difficulty of deciding the degree of size change and motion suitable in every circumstance. For this reason, a language for describing mathematical expressions was designed; this language is translated into **troff** input by a program called **eqn**.

An important requirement is that the language should be easy to learn and use by people who don't know mathematics, computing, or typesetting. This implies that normal mathematical conventions about operator precedence, parentheses, and the like cannot be used, for otherwise the user would have to understand what was being typed. Further, there should be few rules, keywords, and special symbols, and few exceptions to the rules. Finally, standard actions should take place automatically—size and font changes should follow normal mathematical usage without user intervention.

When a document is typed, mathematical expressions are entered as part of the text, but are marked by user-settable delimiters. **eqn** reads the entire input and passes through untouched those parts that are not mathematics. At the same time, it converts the mathematical parts into the necessary **troff** commands. The result is passed to **troff**. UNIX makes it easy to direct the output of one program into the input of another, with a mechanism called a "pipe." Thus normal usage is a pipeline of the form

```
eqn files | troff
```

The two programs run concurrently, with scheduling and input-output synchronization handled by the operating system.

The **eqn** language is defined by a grammar, written in a compiler-compiler language called Yacc,[5] to insure regularity and ease of change. We will not describe **eqn** in detail; see Refs. 6 and 7. Nonetheless, it is worth showing a few examples to give a feeling for the language. Throughout this section we write expressions exactly as they are typed by the user, except that in general we will omit the delimiters that mark the beginning and end of each expression.

eqn is an oral (or perhaps aural) language. To produce

$$2\pi \int \sin(\omega t)dt$$

one writes

```
.EQ
2 pi int sin ( omega t)dt
.EN
```

The markers **.EQ** and **.EN** delimit the expression. They are passed through to **troff**; most macro packages provide definitions for them that cause the equation to be displayed, as it is here, perhaps with a number in the margin.

Each "word" within the expression is looked up in a table. In this case, **pi** and **omega** are recognized as Greek letters, **int** is a special character, and **sin** is to be placed in Roman font instead of italic, following conventional practice. Parentheses and digits are also made Roman, and spacing is adjusted around characters to give a more pleasing appearance.

Subscripts, superscripts, fractions, radicals, and the like are introduced by words used as operators:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r}$$

is produced by

```
x sup 2 over a sup 2 ~=~ sqrt {pz sup 2 + qz + r}
```

Braces { and } are used to group items that are to be treated as a unit, such as all the terms to go under the radical. The precedence of `sup` is higher than that of `over` so braces are not needed for the left side of this expression. **eqn** input is free-form, so blanks and new lines can be used freely to make the input easier to type, read, and subsequently edit. The tilde ~ forces extra space into the output; **eqn** does not automatically add the extra space that would normally appear around the = sign.

The operator `sub` produces a subscript in the same manner as `sup` produces a superscript; if `sub` and `sup` occur together they are treated specially to keep the subscript and superscript nearly vertical, so that

```
x sub i sup 2
```

is printed as x_i^2, not $x_i{}^2$.

```
roman erf (z) ~=~ 2 over sqrt pi int sub 0 sup z e sup -t sup 2 dt
```

produces

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$$

"`roman`" forces the next object into Roman font; it would otherwise be italicized.

Limits may be placed below and above a construct with `from` and `to`:

```
zeta (s) ~=~ sum from k=1 to inf k sup -s ~~~ ( Re ~ s > 1)
```

is

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\mathrm{Re}\ s>1)$$

and

```
lim from {x -> pi /2} ( tan~x) sup {sin~2x} ~=~ 1
```

yields

$$\lim_{x\to\pi/2}(\tan x)^{\sin 2x} = 1$$

Built-up brackets, braces, etc., are available:

```
lim from {x -> inf} left ( 1 + 1 over x right ) sup x ~=~ e
```

produces

$$\lim_{x\to\infty}\left(1+\frac{1}{x}\right)^x = e$$

In such expressions, the character following the keywords `left` or `right` can be any of a family of characters that stretch vertically, such as braces, brackets, bars, floor and ceiling. The character will be stretched enough to cover the enclosed construction.

eqn also provides for matrices; in fact, "matrix" is used in more situations than might be supposed from the name:

$$\begin{aligned} A &= \tfrac{1}{2}\, ab \sin\theta \\ c^2 &= a^2+b^2-2ab\cos\theta \\ S &= \pi r\sqrt{r^2+h^2} \\ V &= \frac{\pi}{3} r^2 h \end{aligned}$$

is produced by a matrix of three columns and four rows:

```
matrix {
   rcol { A above c sup 2 above S above V }
   ccol { = above = above = above = }
   lcol { half ~ ab ~ sin ~ theta
      above a sup 2 + b sup 2 - 2ab ~ cos ~ theta
      above pi r sqrt {r sup 2 + h sup 2}
      above pi over 3 r sup 2 h
   }
}
```

Matrices are specified by columns, each of which may be independently centered or left or right justified.

eqn also provides for diacritical marks such as dots and bars:

```
x dotdot ~=~ y bar
```

produces

$$\ddot{x} = \bar{y}$$

Font and size changes are available to override defaults, as are facilities for lining up equations and a rudimentary macro substitution process.

Because not all potential users have access to a typesetter, there is also a compatible version of **eqn** that interfaces to **nroff** for producing output on terminals capable of half-line motions and printing special characters. The quality of terminal output leaves something to be desired, but it is often adequate for proofreading and some internal uses.

The **eqn** language has proven to be easy to learn and use; at the present time, several hundred typists and secretaries use it at Bell Laboratories. Most are either self-taught, or have learned it as part of a UNIX course taught by other secretaries and typists. Empirically, mathematically trained users (mathematicians, physicists, etc.) can learn enough **eqn** in a few minutes to begin useful work, for its syntax and rules are similar to the way that mathematics is actually spoken. Persons not trained in mathematics take longer to get started, because the language is less familiar, but it is still true that an hour or two of instruction is enough to begin doing useful work.

By intent, **eqn** does not know much about typesetting; in general, it lets **troff** do as much of the job as possible, including all character-width computations. In this way, **eqn** can be relatively independent of the particular character set, and even of the typesetter being used.

The basic design decision to make a separate language and program distinct from **troff** does

have some drawbacks, because it is not easy for **eqn** to make a decision based on the way that **troff** will produce the output. Nonetheless, these drawbacks seem unimportant compared to the benefits of having a language that is easily mastered, and a program that is separate from the main typesetting program. Changes in one program generally do not affect the other; both programs are smaller than they would be if they were combined. And, of course if one doesn't use **eqn**, there is no cost, since **troff** doesn't contain any code for it.

6. TBL—A PREPROCESSOR FOR TABLES

Tables also present typographic difficulties. The primary difficulty is deciding where columns should be placed to accommodate the range of widths of the various table entries. It is even harder to arrange for various lines or boxes to be drawn within the table in a suitable way. **tbl**[8] is a table construction program that is also an independent preprocessor, quite analogous to **eqn**.

tbl simplifies entering tabular data, which may be tedious to type or may be generated by a program, by separating the table format from its contents. Each table specification contains three parts: a set of options affecting the whole table, such as "center" or "box"; then a set of commands describing the format of each line of the table; and finally the table data. Each specification describes the alignment of the fields on a line, so that the description

```
L R R
```

indicates a line with three fields, one left adjusted and two right adjusted. Other kinds of fields are "C" (centered) and "N" (numerical adjustment), with "S" (spanned) used to continue a field across more than one column. For example,

```
C S S
L N N
```

describes a table whose first line is a centered heading spanning three columns; the three columns are left-adjusted, numerically adjusted, and numerically adjusted respectively. If there are more lines of data than of specifications (the normal case), the last specification applies to all remaining data lines.

A sample table in the format above might be

Position of Major Cities		
Tokyo	35°45′ N	139°46′ E
New York	40°43′ N	74°01′ W
London	51°30′ N	0°10′ W
Singapore	1°17′ N	103°51′ E

The input to produce the above table, with tab characters shown by the symbol Ⓣ, is as follows:

```
.TS
center, box;
C S S
L N N.
Position of Major Cities
Tokyo Ⓣ 35°45' N Ⓣ 139°46' E
New York Ⓣ 40°43' N Ⓣ 74°01' W
London Ⓣ 51°30' N Ⓣ 0°10' W
Singapore Ⓣ 1°17' N Ⓣ 103°51' E
.TE
```

Just as in **eqn**, the delimiters **.TS** and **.TE** set off the table specification; they are passed through to **troff** where they may well be interpreted as macros that will perform such operations as keeping the table together on a page, centering it, leaving space around it, etc.

tbl also provides facilities for including blocks of text within a table. A block of text may contain any normal typesetting commands, and may be adjusted and filled as usual. **tbl** will arrange for adequate space to be left for it and will position it correctly. Extensive glossaries and other

items that would often not be thought of as tables are produced this way with our software, to take advantage of **tbl**'s ability to adjust positioning and draw boxes more easily than a typist can. For example, the table on the next page uses text blocks, line and box drawing, size and font changes, and the facility for centering vertical placement of the headings (compare the heading of column 3 with that of columns 1 and 2).

Functional Systems		
Function Number	**Function Type**	**Solution**
1	LINEAR	Systems of equations all of which are linear can be solved by Gaussian elimination.
2	POLYNOMIAL	Depending on the initial guess, Newton's method ($f_{i+1}=f_i-\frac{f_i'}{f_i}$) will often converge on such systems.
3	ALGEBRAIC	The program ZONE by J. L. Blue will solve systems for which an accurate initial guess is not known.

The input for this table is as follows:

```
.TS
center, box;
cB s s
cBp9 | cBp9 | cBp9w(2.2i)
cBp9 | cBp9 | ^
n | ap8 | lp8.
Functional Systems
.sp 3p
=
.sp 3p
Function  Function  Solution
Number    Type
-
.sp 3p
1    LINEAR    T{
.fi
Systems of equations all of which are linear ...
T}
2    POLYNOMIAL     T{
Depending on the initial guess, Newton's method
($f sub i+1 = f sub i - {f sub i '} over {f sub i}$) ...
T}
3    ALGEBRAIC T{
The program ZONE by J. L. Blue will solve ...
T}
.sp 3p
.TE
```

Note that there is no difficulty with equations in tables. In fact, there is sometimes a choice between writing a matrix with the matrix commands of **eqn** or making a table of equations. Typically, the typist picks whichever program is more familiar.

The **tbl** program writes **troff** code as output, just as **eqn** does. This code computes the width of each table entry, decides where to place the columns and lines separating them, and prints the table. **tbl** itself does not understand typesetting: it does not know the widths of characters, and may (in the case of equations in tables) have no knowledge of the height, either. However, it writes **troff** output that computes these sizes, and adjusts the table accordingly. Thus tables can be

printed on any device and in any font without additional work.

As with **eqn**, **tbl** is normally run in a pipeline, perhaps with **eqn** as well:

```
tbl files | eqn | troff
```

Most of the comments about using **eqn** apply to **tbl** as well: it is easy to learn and is in wide use at Bell Laboratories. Since it is a program separate from **troff**, it need not be learned, used, or paid for if no tables are present. Comparatively few users need to know all of the tools: typically, the workload in one area may be mathematical, in another area tabular, and in another only ordinary text.

7. BIBLIOGRAPHIC REFERENCES

Another preprocessor, **refer**, finds and formats bibliographic references. It can be used with a public data base of citations to avoid the need for each author to keep track of precise citations. We have, for example, a list of 5,000 papers in computer science and related areas, any of which can be inserted at any point. To cite the report "Some Applications of Inverted Indexes on UNIX", which describes **refer** in detail, writing

```
.[
lesk inverted indexes
.]
```

produces the citation,[9] suitably formatted, in the text. A few words chosen from author names, title words, or journal names are usually enough to select the correct paper.

The index of citations is maintained using hashed inverted files for fast retrieval. The indexing program extracts the words from the citations and truncates them to six characters (this avoids requiring the retriever to type the exact spelling out to the end). The words are then converted to hash codes, and a conventional inverted file index is made on the hash codes. The size of the hash table is chosen to fit in **refer**'s memory, so that for each word in the citation request (bracketed by **.[** and **.]**) only one seek is made to the disk. The lists of citations for each word in the request are then merged, and the text of the citations retrieved. The program then checks for false drops (relatively rare if several search terms are given) and formats the output for typesetting.

In line with the general **troff** philosophy, **refer** itself delivers a set of string definitions containing the information in the citation: who the author is, what the title is, etc. The formatting macro packages define what the style of a reference is in the particular format being printed. Thus, the reference for the original version of this paper is in the data base as

```
%T UNIX Time-Sharing System: Document Preparation
%K unix bstj
%A B. W. Kernighan
%A M. E. Lesk
%A J. F. Ossanna
%J Bell Sys. Tech. J.
%V 57
%N 6
%P 2115-2135
%D 1978
```

and the final macro package decides whether this will be printed with the article title in quotes or italics, for example. Each macro package for a particular journal style needs as a part, therefore, the description of the bibliographic reference style imposed by that journal.

In addition, the **refer** program can sort the references in an article into alphabetical order by author or title and number them sequentially or by author name and date or other styles. The same data base of papers can also be used to prepare bibliographies and publication lists for individuals. Any number of private and public data bases can be used with **refer**; many of our users maintain private bibliographies for this purpose. In addition, if a reference is not likely to be needed again, the entry can be typed into the text at the appropriate point, and **refer** can be used to format and

sort it, without making use of the data base.

Experience with **refer** has been mixed. The use of the program to format and sort citations is welcomed. The public data base, however, is difficult to maintain because different users have different, and strongly held, ideas about what should be in it. Although these differences are often trivial (should authors' first names be given or only initials; should the *Bell System Technical Journal* be abbreviated as above or as *BSTJ*) they nevertheless often require people to make private bibliography lists rather than using the public list.

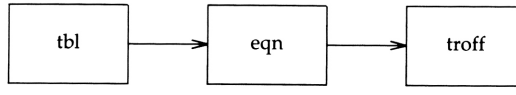
8. GRAPHICS – LINE DRAWINGS

When we acquired our first CRT typesetter, it became possible to make line drawings as well as typeset text. The **troff** program now accepts commands to draw lines at arbitrary angles, instead of just horizontal and vertical lines. For those authors who have slipped from the straight and narrow, there are also commands to draw circular arcs and spline curves.

There are two programs to draw pictures: **pic** and **ideal**. Both are language-based: they use an ordinary terminal as input device, with the picture description spelled out as a sequence of statements in a language. **pic** is like a conventional algorithmic language: the user asks the typesetter to draw this, then to draw that, and so on until the picture is finished. The **ideal** program is analogous to an applicative language: the user states a set of constraints sufficient to define the picture, and **ideal** determines the details and prints the picture. In practice **pic** is more heavily used by typists, since it is easier to understand and somewhat more specialized to the kinds of line drawings customary in our work. We will discuss only **pic** and refer those interested in **ideal** to Chris Van Wyk's paper.[10]

The basic objects in **pic** are boxes, lines, arrows, circles, ellipses, arcs and splines, which may be placed anywhere and labeled with arbitrary text. Normally, these objects can be placed relative to one another, rather than needing absolute coordinates for location.

As a simple example, the figure



is entered as

```
.PS
box "tbl"; arrow; box "eqn"; arrow; box "troff"
.PE
```

The **pic** program is implemented as another preprocessor; as with **eqn**, the language is defined by a Yacc grammar. With a preprocessor, it is much easier to include text, perhaps containing mathematics or tables, inside figures than it would be using a separate co-routine followed by electronic cut-and-paste. Since our local environment until recently had no good graphics input devices, it is natural that **pic** uses ordinary terminals for input; a textual representation also fits naturally with the other components of the system.

The basic elements of **pic** are the objects: box, circle, ellipse, line, spline, ..., and the motions left, right, up, down, ..., which may be combined using conventional programming concepts of nesting and labeling. In addition, there are modifiers of the objects, such as the text inside a box, or the direction of an arrow. Saying

```
box "word"
```

puts the word inside the box, and

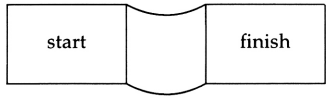
```
arrow right 1i
```

draws an arrow right 1 inch. Normally, a simple list of items is turned into a set of drawings, lined up left to right as in the first example above. However, lines may be drawn between specific

points. Labels can be placed on pieces of a picture, and the corners of these pieces specified with modifiers such as "ne" or "e". Thus, for example,

```
A: box "start"
move
B: box "finish"
arc from A.ne to B.nw
arc from A.se to B.sw
```

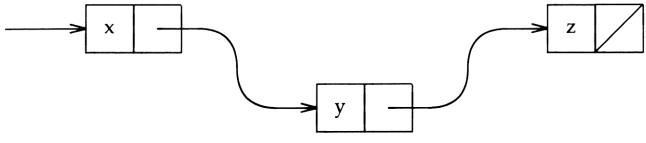
draws



pic also provides the notion of an ordinal name to supplement conventional labels. This example can also be written as

```
box "start"; move; box "finish"
arc from 1st box.ne to last box.nw
arc from 1st box.se to 2nd box.sw
```

The "last" object of a given type is of course the most recent, as illustrated here:



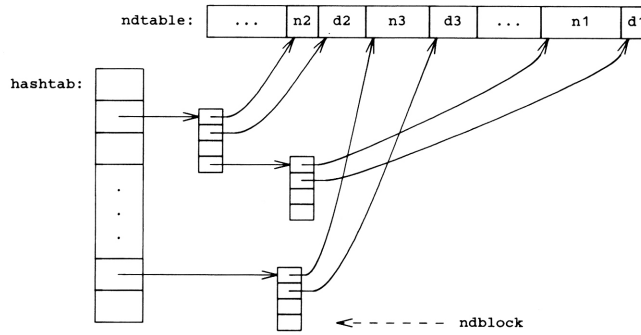
The input for this figure is

```
boxht = 0.3i; boxwid = boxht
arrow
box "x"; box
spline -> from center of last box right then down then right
box "y"; box
spline -> from center of last box right then up then right
box "z"; box
line from last box.sw to last box.ne    # diagonal line
```

The default sizes of objects may be overridden either locally or (as here) by setting the values of standard variables that control them. The curves are quadratic B-splines; the notation -> calls for an arrowhead.

The **pic** language also contains macro expansion, location by x,y position, and invisible objects (used to locate other, visible objects). For further details, see Ref. 11.

As a final, more complex example, here is a chained-overflow hash table:



The problem of creating **pic** input is being addressed with still other preprocessors. Programs exist, for example, to generate **pic** input from the output of a graphics terminal, permitting those with better hardware to take advantage of it. Another program abstracts numerical data from a report and generates the **pic** input to draw a histogram from it. As is typical in the UNIX environment, existing software is used in new ways in preference to writing new packages.

9. OTHER SUPPORTING SOFTWARE

One advantage of doing document preparation in a general-purpose computing environment instead of with a specialized word processing system is that programs not directly related to document preparation may often be used to make the job easier. In this section, we discuss some examples from our experience.

One of the most tedious tasks in document preparation is detection of spelling and typographical errors. Existing data bases originally obtained for other purposes are used by a program called **spell**, which detects potential spelling mistakes. Machine-readable dictionaries (more precisely, word lists) have been available for some time. Ours was originally used for testing hyphenation algorithms and for checking voice synthesizer programs. It was realized, however, that a rudimentary program for detecting spelling mistakes could be made simply by comparing each word in a document with each word in the dictionary; any word in the document but not in the dictionary is a potential misspelling.

The first program for this approach was developed in a few minutes by combining existing UNIX utilities for sorting, comparing, etc. This was sufficiently promising that additional small programs were written to handle inflected forms like plurals and past participles. The resulting program was quite useful, for it provided a good match of computer capabilities to human ones. The machine can reduce a large document to a tractable list of suspicious words that a human can rapidly scan to detect the genuine errors.

Naturally, normal output from **spell** contains not only legitimate errors, but a fair amount of technical jargon and some proper names. The next step is to use that output to refine the dictionary. In fact, we have carried this step to its logical conclusion, by creating a brand new dictionary that contains only words culled from documents. This new dictionary is about one-third the size of the original, and produces better results.

Further checking of documents is possible with the **style** and **diction** programs of Lorinda Cherry.[12] These have been popularized through their inclusion in the Writer's Workbench software package. The **style** program uses a part-of-speech analyzer to determine sentence categories. It measures properties of documents related to readability: Flesch and other readability indices, the number of complex and compound sentences, and so forth. Authors can use this information to revise documents to make them simpler and easier to read. For example, this paper is at 12th grade

reading level and contains 41% simple sentences, 27% complex sentences, 16% compound sentences and 15% compound-complex sentences. If the authors of this paper were less stubborn about taking advice from a computer on how to write, we might revise it to include more short and simple sentences and thus make it easier to read (there is a 55-word sentence in this paper).

Similarly, the **diction** program looks for hackneyed or commonly misused phrases, marking occurrences such as "in the case that" or "sophisticated" in the text, since revision to eliminate these phrases often improves the style of the document. These programs obviously do not represent an alternative to editing and revising, but they do call the attention of authors to things that might be improved. **style** and **diction** have become quite popular, since most authors are interested in writing better.

One of the more interesting peripheral devices supported by the UNIX system is an inexpensive voice synthesizer.[13] The program **speak**[14] uses this synthesizer to pronounce arbitrary text. Speaking text has proven especially handy for proofreading tedious data like lists of numbers: the machine speaks the numbers, while a person reads a list in parallel.

Another example of a borrowed program is **diff**,[15] which compares two inputs and prepares a list of all the places in which they differ. Normally, **diff** is used for comparing two versions of a program, as a check on the changes that have been made. But of course it can also be used on two versions of a document as well. In fact, the **diff** output can be captured and used to produce a set of **troff** commands that will print the new version with marginal bars indicating the places where the document has been changed.

We have already mentioned several major preprocessors for **troff** and **nroff**. The same approach, of writing a separate program instead of cluttering up an existing one, has been applied to *post*processors as well. Typically, these postprocessors are concerned with matching **troff** or **nroff** output with the characteristics of some different output device. One example is a processor called **col** that converts **nroff** output containing reverse motions (e.g., multi-column output) into page images suitable for printing on devices incapable of reverse motion. An analogous program sorts the output of **troff** into vertical order for typesetters that are incapable of reverse paper motion. There are also programs that interpret **troff** output intended for a phototypesetter for display on a terminal. This permits a view of the formatted document without actually printing it, which is convenient for checking page layout.

One final area worth mentioning concerns the problem of training new users. Since there seems to be no substitute for hands-on experience, a program called **learn** was written to walk new users through sets of lessons.[16] Lesson scripts are available for fundamentals of UNIX file handling commands, the editor **ed**, **eqn**, and a standard macro package, as well as for topics not related to document preparation. **learn** has been heavily used in the courses taught by secretaries and typists for their colleagues.

10. EXPERIENCE

UNIX document preparation software has now been used for several years within Bell Laboratories, with many secretaries and typists in technical organizations routinely preparing technical memoranda and papers. Several books[17,18,19,20,21,22,23] printed with this software have been published directly from camera-ready copy. Technical articles have been prepared in camera-ready form for periodicals ranging from the *Journal of the ACM* to *Science*.

The longest-running use of the UNIX system for document preparation is in the Bell Laboratories Legal and Patent Division, where patent applications have been prepared on a UNIX system for a decade. Computer program documentation has been produced for several years by clerks using UNIX facilities at the Business Information Systems Programs area of Bell Laboratories. More recently, the "word processing" centers at Bell Laboratories have begun significant use of the UNIX system because of its ability to handle complicated material effectively.

It can be difficult to evaluate the cost-effectiveness of computer-aided versus manual documentation preparation. We took advantage of the interest of the American Physical Society in the UNIX system to make a systematic comparison of costs of their traditional typewriter composition

and a UNIX document preparation system. Five manuscripts submitted to *Physical Review Letters* were typeset at Bell Laboratories, using the programs described above to handle the text, equations, tables, and special layout of the journal.

On the basis of these experiments, it appears that computerized typesetting of difficult material is substantially cheaper than typewriter composition. The primary cost of page composition is keyboarding, and the aids provided by UNIX software for input of complex mathematical and tabular material reduce input time significantly. Typing and correcting articles on the UNIX system, with an experienced typist, was between 1.5 and 3.3 times as fast as typewriter composition. Over the trial set of manuscripts, input using the UNIX system was 2.4 times as fast. These documents were extremely complicated, with many difficult equations. Typists at *Physical Review* averaged less than four pages per day, whereas our (admittedly proficient) UNIX system typist could type a page in 30 minutes. We estimate a substantial saving in production cost for camera-ready pages using a UNIX system instead of conventional composition or typewriting. A typical UNIX system for photocomposition of *Physical Review* style pages might produce 200 finished pages per day on a capital investment of about $200,000 and with 20 typists. Further details of this comparison can be found in Ref. 24. The American Physical Society in fact has been using a UNIX system for some years now to produce approximately 35000 pages per year of *Physical Review*, a family of heavily mathematical journals. Their staff of typists has been reduced from 38 to 20.

11. CONCLUSIONS

It is important to note that these document preparation programs are simply application programs running on a general-purpose system. Any document preparation user can exercise any command whenever desired.

As mentioned above, a surprising number of the programming utilities are directly or indirectly useful in document preparation. For example, the program that makes cross-reference listings of computer programs is largely identical with the one that makes keyword-in-context indexes of natural language text. It is also easy to use the programming facilities to generate small utilities, such as one that checks the consistency of equation usage.

Besides applying programming utilities to text processing, we also apply document processors to programs and numerical data. Statistical data are often extracted from program output and inserted into documents. Computer programs are often printed in papers and books; because the programs are tested and typeset from the same source file, transcription errors are eliminated.

In addition to the technical advantages of having programming and word processing on the same machine, there can be personnel advantages. The fact that secretaries and typists work on the same system as the authors allows both to share the document preparation job. A document may be typed originally by a secretary, with the author doing the corrections; in the case of an author who types rough drafts but doesn't like editing after proofreading, the reverse may occur. We have observed the full spectrum, from authors who give hand-written material to typists in the traditional manner to those who compose at the terminal and do their own typesetting. Most authors, however, seem to operate somewhere in between.

The availability of the system for electronic mail, calendar service, and other facilities also helps a great deal. It is more convenient to use one system for all these functions than to learn separate command languages for each function, let alone have to use separate terminals and devices. The presence of the secretaries on the same system, for example, makes it easy to send mail back and forth, and most people do this routinely.

The UNIX system provides a convenient and cost-effective environment for document preparation. A first-class program development facility encourages the development of good tools. The ability to use preprocessors has enabled us to write separate languages for mathematics, tables, and several other formatting tasks. The separate programs are easier to learn than if they were all jammed into one package, and are vastly easier to maintain as well. And since all of this takes place within a general-purpose operating system, programs and data can be used as convenient, whether they are intended for document preparation or not.

Acknowledgements

We are indebted to Rich Graveman for his careful reading of the manuscript.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (1978). See ED (I).
2. J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX Programmer's Manual* **2**, Section 22 (January 1979).
3. J. E. Saltzer, "Runoff," in *The Compatible Time-Sharing System,* ed. P. A. Crisman, M.I.T. Press, Cambridge, Mass. (1965).
4. M. D. McIlroy, "The Roff Text Formatter," Computer Center Report MHCC-005, Bell Laboratories, Murray Hill, New Jersey (October 1972).
5. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," *UNIX Programmer's Manual* **2**, Section 19 (January 1979).
6. Brian W. Kernighan and Lorinda L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM* **18**(3), pp. 151-157 (1975).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Comp. Sci. Tech. Rep. No. 17, Bell Laboratories, Murray Hill, N.J. (April 1977).
8. M. E. Lesk, "Tbl — A Program to Format Tables," *UNIX Programmer's Manual* **2**, Section 10 (January 1979).
9. M. E. Lesk, "Some Applications of Inverted Indexes on the UNIX System," *UNIX Programmer's Manual* **2**, Section 11 (January 1979).
10. C. J. Van Wyk, "A Graphics Typesetting Language," *Proc. ACM SIGOA/SIGPLAN Conference on Text Manipulation,* Portland, Oregon (1981).
11. B. W. Kernighan, "PIC – A Language for Typesetting Graphics," *Software – Practice & Experience* (January, 1982).
12. L. L. Cherry, "Writing Tools - The STYLE & Diction Programs," Comp. Sci. Tech. Rep. No. 91 (February 1981).
13. Federal Screw Works, *Votrax ML-1 Multi-Lingual Voice System.*
14. M. D. McIlroy, "Synthetic English Speech by Rule," Comp. Sci. Tech. Rep. No. 14, Bell Laboratories, Murray Hill, New Jersey (March 1974).
15. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey (June 1976).
16. B. W. Kernighan and M. E. Lesk, *Computer Aided Instruction on UNIX,* Bell Laboratories internal memorandum (1976).
17. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style, 2nd Edition,* McGraw Hill, New York, New York (1978).
18. C. H. Sequin and M. F. Tompsett, *Charge Transfer Devices,* Academic Press, New York (1975).
19. B. W. Kernighan and P. J. Plauger, *Software Tools in Pascal,* Addison-Wesley, Reading, Mass. (1981).
20. T. A. Dolotta et al., *Data Processing in 1980-1985: A Study of Potential Limitations to Progress,* Wiley-Interscience, New York (1976).
21. Alfred V. Aho and Jeffery D. Ullman, *Principles of Compiler Design,* Addison-Wesley (1977).
22. Committee on Impacts of Stratospheric Change, *Halocarbons: Environmental Effects of Chlorofluoromethane Release,* National Academy of Sciences, Washington, D. C. (1977).
23. W. H. Williams, *A Sampler on Sampling,* John Wiley & Sons, New York (March 27, 1978).

24. M. E. Lesk and B. W. Kernighan, "Computer Typesetting of Technical Journals on UNIX," *Proc. AFIPS NCC* **46**, pp. 879-888 (1977).